APPENDIX A

NVSI VIRTUAL MACHINE TECHNICAL MANUAL

VERSION 3.0 REV MARCH 2002

(24 pages)

Veriscape Technical Report
NVSI 3
November 21, 2000

NVSI Virtual Machine Technical Manual

Preface

NVSI (*Netcentric Virtual Supercomputing Infrastructure*) is a technology that provides a software solution for a broad range of computationally demanding problems – in a myriad of commercial domains – that would normally require a dedicated supercomputer or large-scale, special-purpose, 'one-off' software. The core of NVSI is a system architecture that blends key ideas in computing, some rather novel, and some imported and modified from discrete mathematics, analog and digital computing theory, high-performance computing, and hardware supercomputers. The most unique aspect of NVSI is the design philosophy, derived from a biological perspective, in which analogs of evolution and growth provide computational structures and representations that are dynamic, flexible, adaptive, and work "well enough" under the constraints of imperfect and incomplete information.

As analogy, the human brain is not the ideal solution that a computer engineer would construct from scratch. The brain is, in effect, a hodge-podge of accumulated structures and algorithms that includes the biological equivalent of dead, buggy, bloated, contradictory, redundant, expensive, unstructured, inelegant, non-optimal and apparently useless code, and yet it solves an enormous range of changing situational problems. It doesn't always, or even usually, provide perfect results in an ideal sense, but the brain does yield workable and often novel moment-to-moment solutions that accomplish the task at hand.

Fundamentally, NVSI is a substrate that embeds biological notions in a system architecture, a *virtual* machine enabling construction of high-performance applications that provide superior solutions to certain classes of problems arising in a variety of industries. In particular, NVSI creates a common computing foundation that applies to widely disparate domains, where the shared theme is the rapid response to complex and fluid user demands not requiring the real-time analysis of a massive flood of streaming data, and for which the solutions may be imprecise and incomplete, but sufficiently accurate and robust to relieve the burden on brute-force processing, and to dramatically enhance the responsiveness of the system.

Some typical domains of application include:

- financial risk-analysis & portfolio valuation
- e-commerce procurement & catalog management
- customer, vendor and peer-to-peer knowledge capture
- telecommunications call-routing
- high-demand query caching
- customized global media distribution, and
- fraud detection.

1. NVSI Features

The innovations of NVSI, beyond the leading one of creating a virtual supercomputer in software, include the merging of existing high-performance computing techniques and designs with the employment of novel approaches to computation, such as the separation of pre-computation from navigation to dramatically reduce real-time overhead, biologically-inspired "good enough" solutions, and the use of evolving data structures and adaptive computational architecture. In particular, key aspects of NVSI are:

- **matching problem architecture** – NVSI is designed at every level to enable the architecture of the data space to reflect, or 'map', the architecture of the problem domain. This allows for the most efficient solution to the problem.

- **adaptable solution manifold** – Flexible connectivity in data-structures allows for the optimal hyperspatial topology (or a mosaic of topologies) – selected from a spectrum of representations (such as point-clouds, graphs, trees, lattices, and hypercubes) – that are most relevant to a specified problem domain. Further, the manifold may adapt in a variety of ways, including value interpolation, function extrapolation, and tree elaboration.

- **fast application creation** – The NVSI virtual machine is a unified computational substrate, including not only the virtual "hardware", but a collection of dedicated engines (*configuration, instantiation, population, navigation, evolution*), managers (*network, thread, data-distribution, multiprocessing*), and toolbox functions, which together allow for rapid development of new applications in different domains, as the structure of the problem changes, without having to build a new special-purpose architecture from scratch.

- **evolving structure** – The approach is organic, as NVSI dynamically alters its data-structures and solution-space (manifold) architecture and topology, and even reconfigures its processor design, in response to on-going changes and demands in the problem space. That is, NVSI enables applications to accumulate, select or extinguish the adaptations it makes to variation in both the content and the character of the data. Thus, both the virtual machine, and the domain applications, *evolve*. And while a few commercial supercomputer designs employ reconfigurable computing, they are necessarily limited by realization in hardware, whereas NVSI, being a virtual machine in software, provides nearly unbounded reconfigurability without the burden of acquiring an expensive and exotic mainframe.

- **optimized calculation** – Highly-optimized function evaluation, fixed-point integer arithmetic, application-selectable precision, and various other numerical techniques provide for ultra-fast, 'accurate enough' computation.

- **supercomputer techniques** – Software emulation of high-performance computing structures and processes (such as a small instruction set, simple and efficient data representation and handling, inherent vector representation, limited data/calculation modes, interleaved memory, table lookup, induced pointers, and distributed & parallelized computation) provide a powerful machine *and* cost-effective scaling and enhancement.

- **pre-compute & navigate** – Separation of processes into pre-computation (populating the state-space) and interactive navigation (searching, modifying and selecting the resulting hyperspace of results) allows for near-real-time response, despite highly complex and computationally-intensive data manifolds.

- **autonomous dæmons** – Second-order dedicated processes operate in background, as concurrent tasks, to collect garbage, prune trees, condense redundancies, process edit-queues, interpolate with finer granularity (mesh enhancement) around selected nodes in state-space, or to extrapolate and elaborate the data structures, during both population and navigation phases.

- **integrated multiprocessing** – In addition to its embedded netcentric design optimized for distributed processing, the NVSI virtual machine provides for inherent parallelism as multiple program threads generate multiple CPU clones, an approach not possible with a hardware machine.

2. Design Overview

WHY NVSI?

NVSI is a *virtual* computer, composed of an operating system and a quasi-general-purpose, reconfigurable, multiprocessing, network-based, asynchronous RISC machine, that happens to be created in software only, and is designed to provide high performance over a large class of specific business-computing problems. NVSI can be implemented on a mosaic of platforms, and an increase in power of the underlying physical CPUs simply enhances the overall performance of the system.

One may ask: how can a layer of software be more powerful than writing code directly to the underlying physical platforms? The answer, of course, is that fundamentally, it can not. However, by acting, essentially, as an alternative computing architecture, NVSI is designed to expose to the system developer a landscape of primitive (that is, elementary) operations essential to the kinds of problems for which the features of NVSI are optimum. Thus, applications can be quickly developed for the relevant domains, without a business having to employ extensive labor resources to create a special-purpose, dedicated 'one-off' solution at enormous cost. Further, the internal NVSI operations use as few and the fastest platform-CPU instructions as possible, thus maximizing performance and avoiding the 'bloat' and wasted power inherent in general-purpose microprocessors. And last, NVSI bypasses the platform OS and makes direct calls to the platform CPU, thus eliminating one software layer that applications would typically be forced to use.

The result is that NVSI squeezes maximum performance out of the underlying CPUs, while providing to the developer a set of operations optimally suited to the problem at hand. With computational *power* proportional not only to speed and storage, but also to *utility*, and inversely proportional to capital and human *resources* consumed, NVSI outperforms both hardware supercomputers and custom vertical applications in power per dollar, when applied to the problems for which it is designed.

NVSI ARCHITECTURE

As in any computer, there are CPUs that include functional units, memory stores, and dedicated registers, all of which execute machine-instructions coded by system programmers. The core of the NVSI System is composed of the NVSI Virtual Machine (VM, the 'hardware') and the NVSI Operating System (OS). The NVSI-OS is an integrated collection of managers, engines and Toolbox macros, all written in an NVSI control language (NVCL), which is assembly-like and makes direct calls to the virtual hardware.

A typical system configuration for an NVSI installation is shown in Figure 1, which illustrates one NVSI System installed on one physical machine (server platform). The actual NVSI processor (VM) is shown in more detail in the functional block diagram of Figure 2.

Terminology

Solution Space – the collection of points, or *nodes*, in a state-space representing the entire landscape of data structures that is allocated (*instantiated*), computed (*populated*), explored (*navigated*) and modified (*evolved*). The atomic element of the solution space is a *node*, which represents a point in the solution space to which is attached a set of data structures. The way in which nodes are connected (or not) determines the topology of the solution space (sometimes termed a *manifold*), and this topology, together with the associated node data-structures, implements a "map" of the problem domain. A spectrum of basic topologies for the solution space is briefly described in Appendix I, with more detail provided in the *NVSI System Developer Manual.*

Node – the elemental entity in a solution space. Each node is represented by two parts: an Index Word (IW) that corresponds to the node-number, and includes a pointer to the associated Data Word (DW), which is the second part that comprises a node.[1] The DW is an array of fields that can be of various data-structure types, including numeric or character tags, boolean flags, numeric or character values, arrays, connection pointers (to other nodes), function pointers, lookup-table list pointers, linked-lists, or even pointers to entire other manifolds. There can be many different kinds of data structures in a solution space, and each distinct data-word architecture is defined and stored in the Solution-Space Configuration Unit (SCU, see below).

Unit – a functional component of the VM that performs a specific category of operations on nodes. Some typical operations are: configuring (and reconfiguring) the NVSI architecture for the problem domain; allocating & configuring the index and data spaces; creating, deleting, & populating (evaluating) nodes; navigating (exploring) the node data-space (solution space, or manifold); modifying (evolving) nodes and/or connections; performing optimized arithmetic, getting lookup-table values for common mathematical operations, and generating function values; and distributing data and/or tasks over a network.

Register – an internal part of a Unit, used to transfer data between memory and NVCL variables. The configuration of each register corresponds to an associated memory word, and is thus implicitly composed of sequential fields of specifiable length. Each field in a register is ultimately bound one-to-one to a specific, persistent variable – appropriately named to reflect the field description – in the relevant engine (NVCL) program. However, an important design aspect of the VM 'hardware' is that, to maintain the utility (that is, fastest possible performance) of loading a register in a real machine – which is a parallel operation and thus takes only one hardware cycle – the actual transfer of contents between registers and Index/Data Memory is done as a single binary encoding of the entire word. That is, when NVSI is running in 'compact' mode, only bit-strings are loaded into, and from, Index and Data registers, and are stored into or read from Index and Data memory words. Thus, a register is loaded from external operands by first encoding the field-variables into binary, and then compacting the bits into a single register-operand. Conversely, to extract the individual

[1] For some applications, an Index Word can serve as the Data Word as well, in which case, a node is then represented by only Index Words that also contain some data fields.

fields into NVCL bound-variables, the register contents are assigned to a whole-register variable in NVCL, and this bit-string is then parsed by the program code into the corresponding variables.[2]

The functional units of the VM are all dedicated to manipulating nodes: their data architecture and their connection topology. Each unit is controlled directly by the corresponding engine(s) in the OS, which send instructions to the unit. The NVSI-OS is both netcentric and multitasking, and thus machine programs can not only be sent to multiple VM-CPUs, but can also execute in each of the functional units of a CPU separately, in tandem (see *NVSI Multiprocessing*).

VM Unit Descriptions

The units and their basic functions follow. **Boldfaced** terms in the text refer to other units. *Italicized* terms refer to engines or managers in the NVSI-OS. For reference, the NVSI Register & Memory-Word Specifications are shown in Figure 3, and the NVSI Machine Instruction Set is shown in Table 3 (both are discussed in more detail in later sections).

- **Solution-Space Configuration Unit** (SCU)
  The SCU specifies all of the elements that determine the architecture of the virtual machine, including: the amount of memory allocated for the index and data spaces, the register and field configuration parameters (in the PCR), the array of data-definition words (DDA) that define the architecture of both index and data words & registers, the table of field types, the size of the **ALU** stack, and the memory segment-addressing registers The SCU also stores a dynamic count of the number of nodes instantiated (NNR), and the number of data-word definitions currently active (DDCR). The SCU gets instructions from the *Configuration Engine.*

- **Instantiation Unit** (IU)
  The IU creates and deletes nodes. A node is created by storing a node Index Word (IW) into the **Index Memory** (IM) – at a location given by the **Node Counter** (NC) – with the contents of the Index Word Register (IWR). The IW contains a flag indicating the 'null' (free, available for assignment) status of the node, a pointer to the data-definition word (DDW) for the node, a pointer to the virtual-node-address (VNA) of the node data-word (DW), and one or more application-definable fields (ADF). Space for the DW is reserved in the **Data Memory** (DM) at a location given by the VNA, based upon the length specified by the DDW. The actual physical memory location is maintained by the **Physical Memory Controller** (PMC) unit.
  The IU gets instructions from the *Instantiation Engine* and the *Evolution Engine.*

- **Population Unit** (PU)
  The PU stores data into nodes. That is, it fills and/or computes the value of all fields of the node DW – in the manner specified by its DDW – with the data contained in the PU Data Word Register (DWR-P). The address of the DW is the VNA contained in the corresponding IW(NC). Any functions called for by the node DDW (that are to be computed during population) are evaluated and the result stored in the corresponding DW fields. The PU gets instructions from the *Population Engine* and the *Evolution Engine.*

[2] The VM can also be run in 'non-compact' mode – typically for creating application prototypes – in which registers and memory words are simply stored as mixed-type arrays of fields, and thus no binary encoding/extraction is performed. Of course, this severely degrades performance (in both space and time), but it does allow for an easier development process.

- **Node-Counter Register Unit** (NC)
  The NC holds the node-number, an integer that points to a word in the node **Index Memory** (IM), with the pointer-value an integer in the range [*0* : *maxNodeNum*]. The NC is used by the **IU**, **PU** and the **Navigation Unit** to select the node to operate upon. The NC may get instructions from the *Instantiation, Population, Navigation* and *Evolution* engines.

- **Navigation Unit** (NU)
  The NU reads a selected node. Given the node number (in the **NC**), the data-word (DW) is read from the data-memory – at the VNA in the corresponding index word – and loaded into a register (DWR-N) in the NU. If the node specification requires navigation-time evaluation of functions, this is performed by the *Navigation Engine* program after the data word is read.
  Typical navigation often involves moving among connected nodes. This is accomplished by reading a DW, extracting the node-number pointer from the relevant data field, executing an instruction to set the **NC** to that value, and then executing a 'read' again.
  The NU gets instructions from the *Navigation Engine.*

- **Index Memory** (IM)
  This is a fast, random-access, shared memory store that contains the node index table, which is an array of index-words for the entire data space. The index-word (IW) address is the node number, which is taken from the **NC** (or the NNR). The computation of the VNA for the data-word (DW) is handled internally (and automatically) by the data-memory-manager (DMM) when a node is created (via the **IU**), using the contents of the **NC** and the corresponding DDW. The VNA is output to the **Physical Memory Controller** (PMC) unit, which handles the actual storage of the associated DW. If a population operation causes the size of a (variable-length) data-word to change, the DMM finds the next available VNA, then updates the VNA field of the corresponding IW, and it also triggers a reload of the IW register in the **IU** (to reflect the new VNA).
  The IM also contains the Node-Free Index (NFI), which is simply a memory-list of all currently available (free) node-number ranges. The IM internal controller maintains the NFI, updating it for every create-node or delete-node operation. In response to a get-free-node-range instruction, the IM controller loads the inclusive bounds of the next (*i.e.,* at or beyond the value of the NC) free-node range into the NFNR1 & NFNR2 registers. This operation is the only significant piece of 'firmware' (that is, embedded microcode program) in the VM (except for some **ALU** routines).
  In a multiprocessing environment, the IM controller also handles the memory sharing. For high-performance applications, the IM is physically located in one platform RAM, although it can, in fact, be spread across networked resources.

- **Data Memory** (DM)
  This is the unit that implements the data manifold. It is a fast, random-access, shared memory store that contains the entire collection of node data-words. The separation of index from data allows the DM to be maximally compact. Indeed, for hypercube-type data spaces, the geometry of the manifold is explicitly stored in the Index. The DM is a virtual memory that is implemented physically over possibly many networked machine RAMs and/or hard-storage devices. The physical node address (PNA) is taken from the **PMC** unit. In a multiprocessing environment, the DM internal memory-manager also handles the memory sharing. Typically, in all but the smaller domain applications, the DM will be spread across networked hard-storage resources. To maintain optimum performance, the data memory is compacted and stored sequentially in the most efficient manner possible (such as disk-striping).

- **Arithmetic & Logic Unit** (ALU)
  The ALU is a 'firmware' emulation of a typical ALU, but of more utility is the built-in support for fixed-length representation of real numbers in one-byte increments, and the use of look-up tables for common mathematical functions, to allow for fast, moderate-accuracy computation. The ALU also stores application-defined functions (such as pointer-induction computation) for access by other units (notably the **PU** and the **NU**). As the underlying platform-CPU may be quite powerful arithmetically, thus negating any advantage of the NVSI ALU for standard arithmetic operations, a set of instruction flags (operands) allow for pass-thru of any or all operations to the platform ALU. The NVSI ALU gets instructions from the *Population Engine* and, for secondary processing where necessary, the *Navigation Engine.*

- **Physical Memory Controller** (PMC)
  The PMC handles the conversion of virtual to real memory addresses, for both RAM and hard storage.

- **Network Control Unit** (NCU)
  The NCU handles the distribution of both data and task processes for the network. It gets instructions from the *Network Manager.*

NVSI Multiprocessing

Multiprocessing is built into the NVSI architecture on several levels:

- There can be one or more NVSI Virtual Machines installed on a given platform (although at most one platform is dedicated to one VM; that is, to conserve performance, a VM is almost never spread over separate physical machines). More typically, a set of machines, each with an NVSI-VM installed, can be networked to provide either for concurrent execution of different applications, or, more powerfully, for the magnified power inherent in multiple machines processing over the same application domain. A network of cooperating VMs is termed an *NVSI Cluster.*

- The NVSI-OS is multi-tasking, so that in a given VM, separate threads may be created and execute concurrently (this is, of course, pseudo-multitasking unless the physical platform – the installation machine – is a true multiprocessor). As each unit in the VM may execute code separately and concurrently, there is no central control unit to act as a bottleneck, which otherwise happens in a conventional Von Neumann machine.

- For each OS thread, a separate, temporary NVSI VM-CPU may be spawned, or replicated. The *core* of the VM (the central 'spine' of Figure 2) contains the global, non-replicating elements (SCU, IM, DM, PMC, NCU) of the machine (one core per VM installation). A replicable virtual CPU (which may be partial) is thus composed of one or more of the following units: IU, NC, PU, NU, & ALU. Local data is modifiable, but all virtual CPUs must negotiate with the SCU and memory units for access to shared information. Note that, implicit within each global unit is the ability to create and manage multiple input data paths, and multiple output buses, to accommodate the multiple virtual-CPUs created by each thread. A partial CPU allows for the creation of simply one or two Units for a thread, as needed.

- The PMC & NCU, controlled by the *Network Manager*, allow for distributed processing and data-distribution across a heterogeneous network of platforms. This is the more conventional use of networked resources.

OS Engine Descriptions

The various engines are low-level modules in the NVSI-OS that generate the actual machine code sent to and executed by the Units of the VM. Thus, the Instantiation Engine, for example, runs procedures to create nodes, the Population Engine procedures fill (evaluate & store) nodes, and so forth. Calls to the engines are made from the OS managers (or perhaps the application directly) as *tasks*, such as "Instantiate 500 nodes", with associated parameters passed to the engines. Some typical engine procedures are shown in Appendix II.

The Toolbox is simply a collection of generic engine programs that have wide utility, both for low-level OS tasks, and application-level functions that are generic across domains. The Toolbox enlarges as experience with various application domains yields engine code that is used repeatedly.

Daemons are autonomous programs used for concurrent, dedicated processes that operate in the background to collect garbage, prune trees, condense redundancies, process edit-queues, etc.

The engines, toolbox and daemons are presented in detail in the *NVSI System Developer Manual.*

Manager Descriptions

While the Engines implement various tasks, the NVSI-OS Managers coordinate among tasks. These include *network, thread, data-distribution,* & *multiprocessing* managers, as well as *engine* managers that handle the coordination of all the different tasks for a particular engine, in a higher-level manner accessible to the NVSI API. The managers are presented in detail in the *NVSI System Developer Manual.*

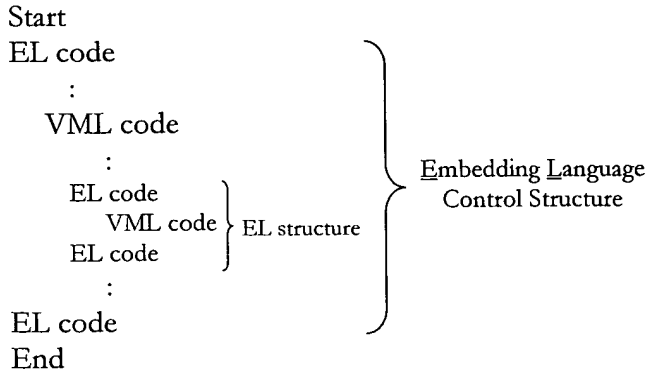
3. Writing NVSI Programs with NVCL

NVSI Control Language (NVCL) is the assembly-level language composed of two coding schemes:

- a *Virtual Machine Language* (VML) that maps one-to-one from the machine instructions (Table 3) to actions by the VM 'hardware' units, and
- an *embedding language* (EL) that handles the flow of control (there are no branch or halt instructions in the NVSI instruction set).

OS engine and/or manager programs are written in NVCL. In subsequent versions of NVSI, there may be provisions for each unit to maintain its own program counter, and thus its own flow-of-control.

A typical engine program in NVCL has the following structure:



NVCL Examples

For all example code-fragments, the Embedding Language (EL) is a form of pseudo-code, in which **boldface** indicates a keyword, *italic* denotes a variable name, and *underline* indicates a named constant. For virtual-machine instructions (shown in monotype), operands appear in sequence after the operation, separated by spaces. 'True' values are coded as '1', 'False' as '0'. A 'null' value codes a non-operative variable or operand.

Figure 1. ***NVSI System: Configuration Diagram***

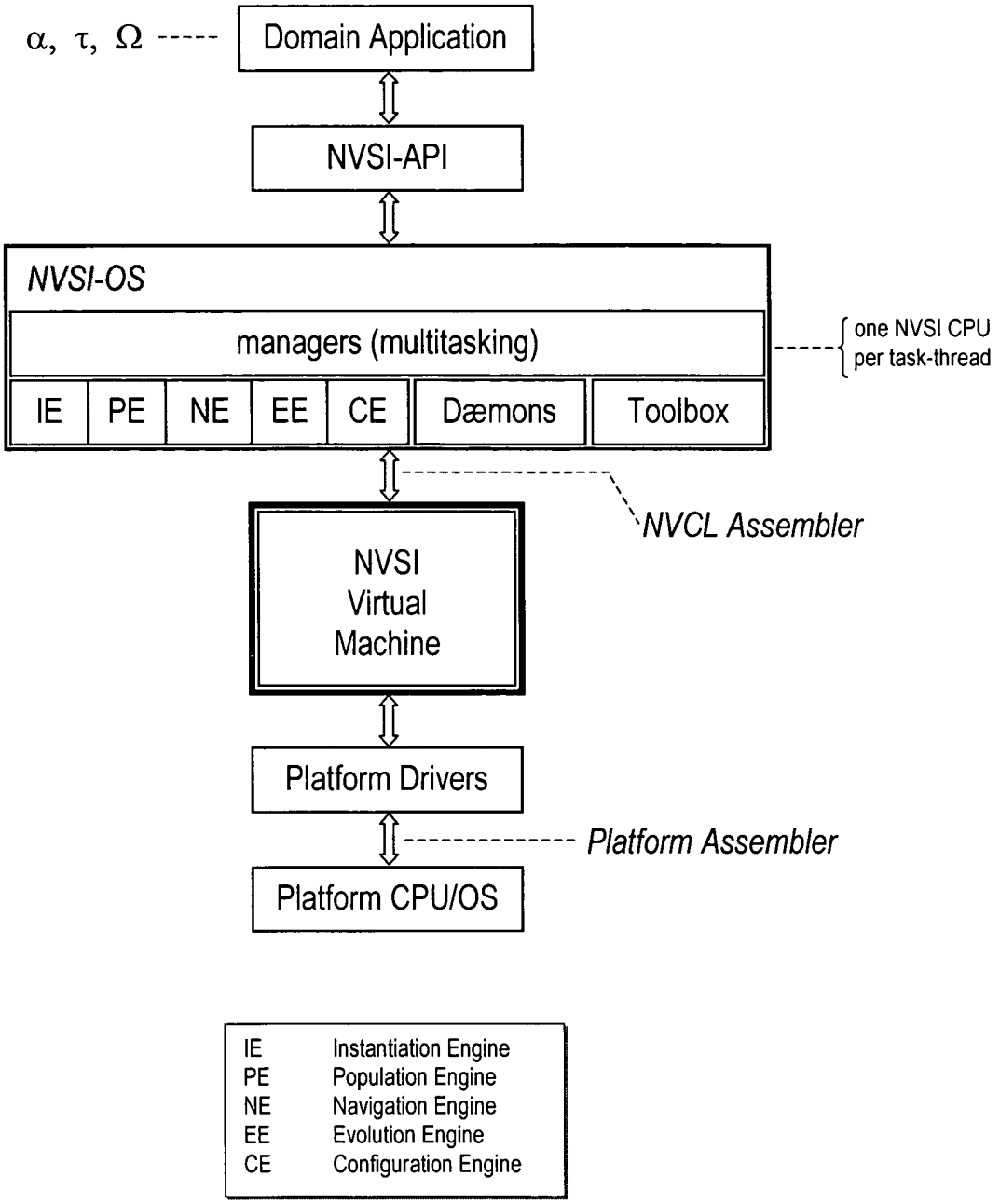


Figure 2. ***Virtual Machine: Functional Block Diagram***

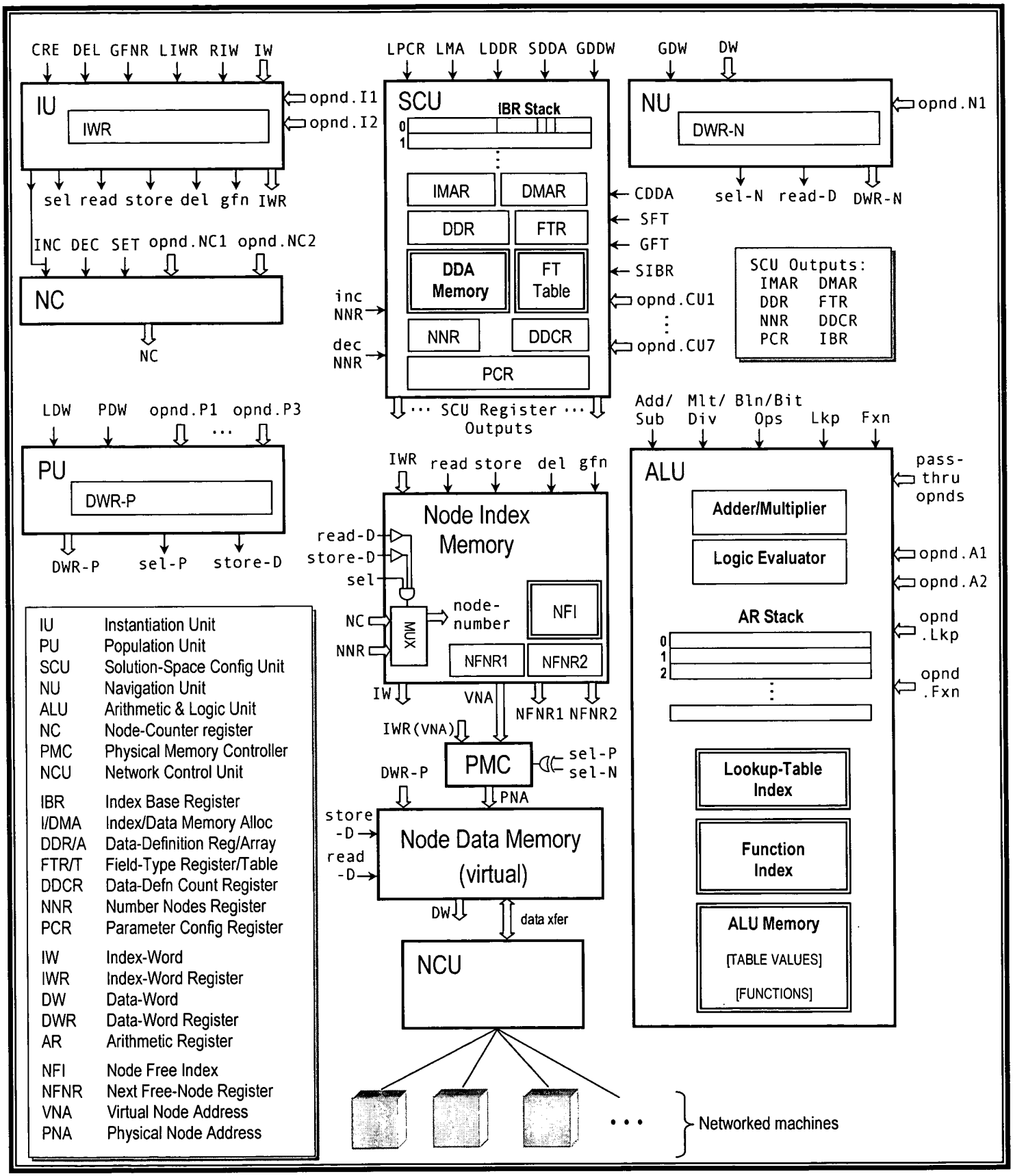
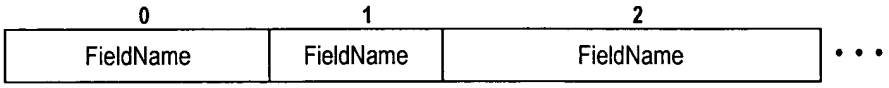


Figure 3. ***Virtual Machine: Register and Memory-Word Specifications***

All word/register configurations are illustrated as follows:

Fields:



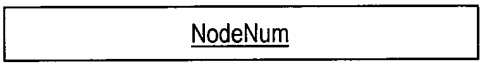
Field numbering begins with left-zero, and is not shown except where necessary to avoid ambiguity. Underlined field names indicate fields that also specify field-type or length in the same or another register. If a field is enclosed in square brackets [...], it is an optional part of a word or register, contingent upon parameters stored in another register or memory word. Field names shown in the text are delimited by angle brackets < ... > .

Field format: Boolean (bln) | integer (int) | character (char)

These are the primary formats for word/register content. There are, however, many field meanings, or *types*, such as floating-point, pointer, etc, that further define the field content, and are specifiable by <TypeNum> fields in a Data Definition Word (DDW). The list of available types is maintained in the Field-Type Table (FTT). The int format may be decimal or binary, and the char may be typographic or binary ASCII, depending on a parameter (*compact mode*) in the Parameter Configuration Register (PCR), or unless noted otherwise.

Field length: **n**, the number of places – bits or bytes, depending on mode.
Unless noted otherwise, **n** is shown in bits . If the length is derived from a parameter stored elsewhere (usually in a definition word or in the PCR register), then it is shown as either **len<parameter>**, or **len(<parameter>)** if **len** = log_2(**parameter**).

Node Counter Register (NC)



int

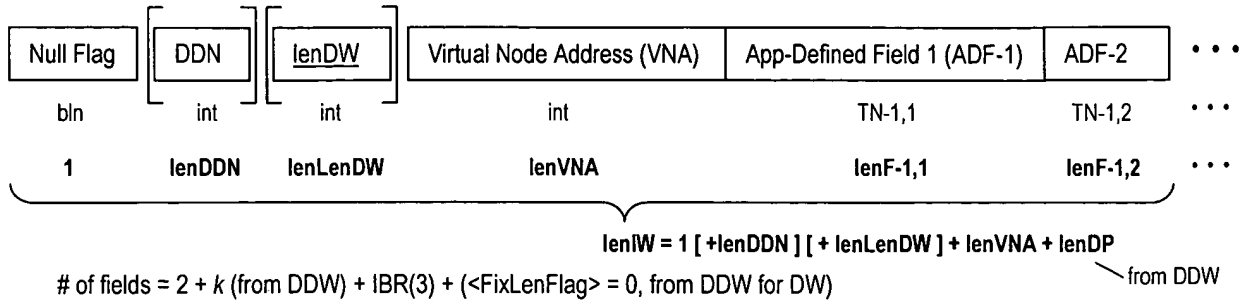
lenNC

The NC is the node-pointer. Its contents are the node-number *NodeNum*, an integer that is also the relative address (starting at zero) of the corresponding node Index Word (IW) in the Index Memory (IM). Thus,

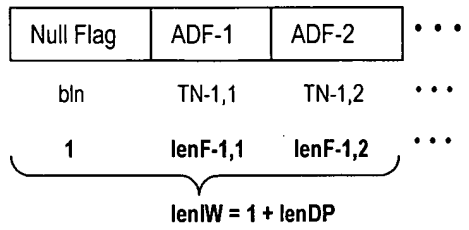
$0 \leq$ *NodeNum* $\leq \Omega$, $\Omega \equiv$ maximum number of nodes = *maxNodeNum*.

Index-Word (IW)

index version



data version



<*Name-α,β*> ≡ field-parameter *β* of data-definition packet *α*, $1 \le \alpha \le j$, $1 \le \beta \le k$ (see DDW).

# of fields = 1 + *k* (from DDW)

The IW typically functions as the index (*index version)* for an associated data word (DW), but the IW can be used as an indexed data-word itself (*data version*), for optimal performance where the data structures are fixed-length and simple.

An IW is always fixed-length, and all words in a memory segment are the same length. This allows for RAM-like storage and retrieval of index words. The address of each IW is its node-number (*NodeNum*).

The <Null Flag> is set only via a DEL instruction. It is redundant, as the null status of any node can be extracted from the Node-Free Index (NFI), but the flag provides faster access to null status at minimal storage penalty.

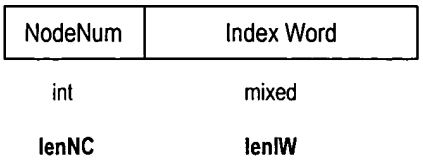
The optional <DDN> field specifies the data-definition number for the associated data word. It exits if the <DDN-D Flag> of the corresponding Index Base-address Register (IBR) is *1*, indicating that the data-definition number is specified in the IW instead of by the <DDN-D> field of the IBR.

The optional <lenDW> field defines the length (in bytes) of the associated data-word (and DW register) for the index version of the IW. It exists if the data-word is variable length, as specified by a *0* value in the <FixLenFlag> field of the data-definition for the data word (available from the DDR register in the SCU). Otherwise, the length is specified in the DDW.

The composition of the Application-Defined Fields in the IW is contained in the corresponding Data-Definition Word (DDW), which is specified in the relevant IBR for the index-memory segment that includes the given IW (addressed by node-number). The DDW is selected by its Data-Definition Number (DDN), and the default DDW that specifies all index-words is DDN *1*.

There are a fixed number of ADFs for both versions of the IW, and all fields are fixed-length. Thus, for a DDW that specifies the configuration of an index word, there is only one definition-packet (*DefPak* – see DDW word, below).

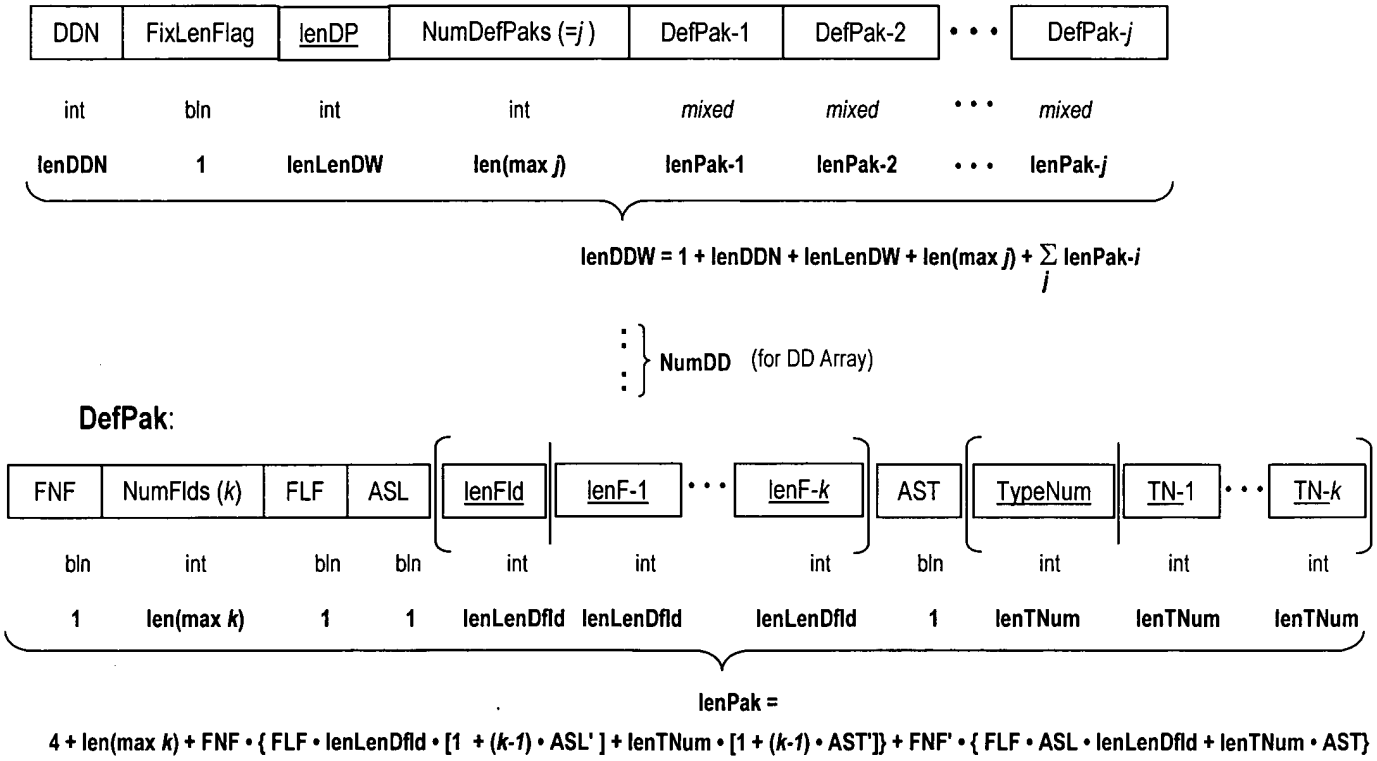
Index-Word Register (IWR)



The IWR holds the contents of one index word, to be either stored into or read from the IW pointed to by NC. The <NodeNum> field is valid only after a read operation, and serves as a memory-address register to retain the last node-number accessed.

In 'compact' mode, the contents are loaded as one binary super-field (operand), either as read from memory, or as constructed from concatenating and encoding all component operands of the embedding-language variables into one bit-string. The parsing of the index-word after a read is done at the engine level, using the specifications contained in the corresponding DDW.

Data-Definition Word (DDW) & Register (DDR)



A DDW specifies in detail the architecture of any data words, and/or the data-portion of the index words. The selector is the Data Definition Number <DDN>. The DDN is not a relative address into the array, it is merely an application-defined integer label (although it is unique). This allows for particular DDNs to be retained for corresponding DDWs, without the need for reassignment and/or word-swapping in the array. The actual addressing of array words is handled internally by the SCU.

When a DDN is specified (≥ *1*) in a data-definition loaded into the DDR, and a store-data-definition instruction is executed, the internal controller checks to see if the DDN is new. If so, the DDW is written over a null array word (an unused, or available, memory word, indicated by DDN = *0*). If the specified DDN is not new, the existing DDW word with the same DDN is overwritten by the contents of the DDR.

The <FixLenFlag> specifies that the entire data-portion (DP) of the definition is fixed-length, and the length is then contained in the <lenDP> field. Although this information could be derived from real-time extraction of the relevant fields in the definition word, these two fields provide fast access for fixed-length configurations. If the DDW applies to an Index-Word (IW) – which are, by definition, fixed-length – then the <FixLenFlag> field is ignored.

If <FixLenFlag> is *0* (variable-length), then the length of each data-word is computed during instantiation or population, and stored in the <lenDW> field of its corresponding IW.

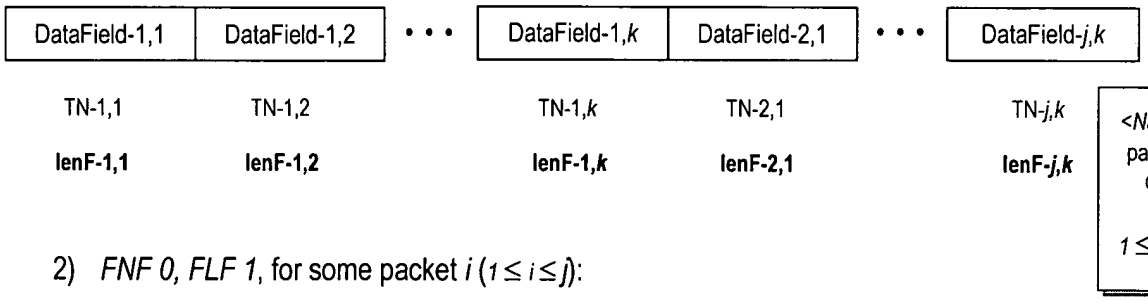
The details of the data-definition are contained in some number *j* (from the <NumDefPaks> field)of *definition packet* (DefPak) meta-fields, which are collections of field-specifications for each packet. If the DDW applies to an Index-Word, there is only one DefPak.

Each DefPak specifies whether or not there a fixed number of fields (<FNF> = *1*), and if so, the number (*k*) of fields is given by <NumFlds>. If all the fields in the packet are fixed-length (<FLF> = *1*), then the length is given by the <lenF-*k*> fields. And if all the fields are the *same* length (<ASL> = *1*), then the length is given by just the one <lenFld> field. If the number of fields in the packet is variable (<FNF> = *0*), then the field lengths are stored in each data-word. Note that if the global FixLenFlag = *1*, then all <FNF> and <FLF> fields are taken to be *1*.

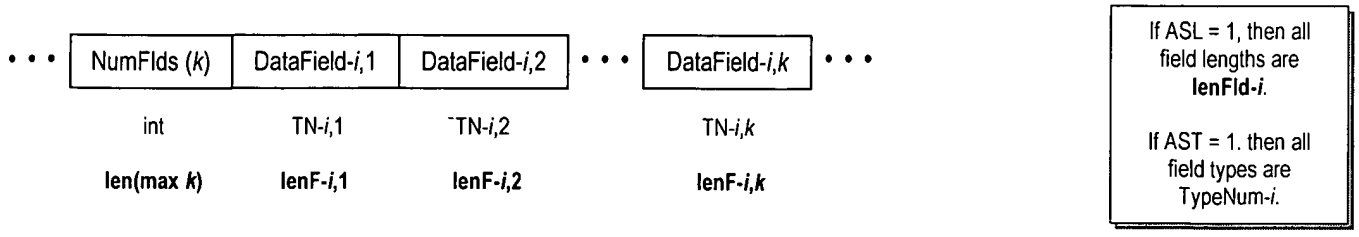
Similarly, the data-field types are given by the <TN-*k*> fields, where field type includes: tag, flag, character, integer, fixed-point, floating-point, function-pointer, node-pointer, and list-pointer (the table of data types is stored in the Field-Type Table in the SCU). And if all the fields are the *same* type (<AST> = *1*), then the type is given by just the one <TypeNum> field. The case where the number of fields is variable (<FNF> = *0*) *and* the fields are not all the same type, is not allowed, as this would force the storage of field types in the data word, which is a complexity not part of NVSI version 1.0.

Data-Word (DW)

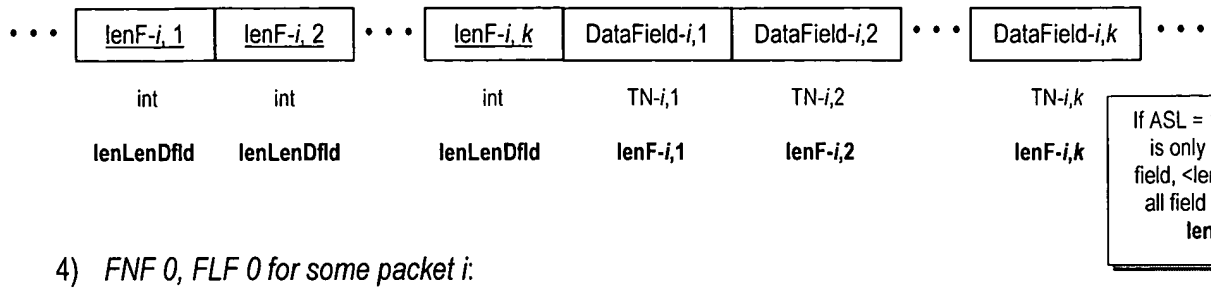
1) *FixLenFlag 1*:



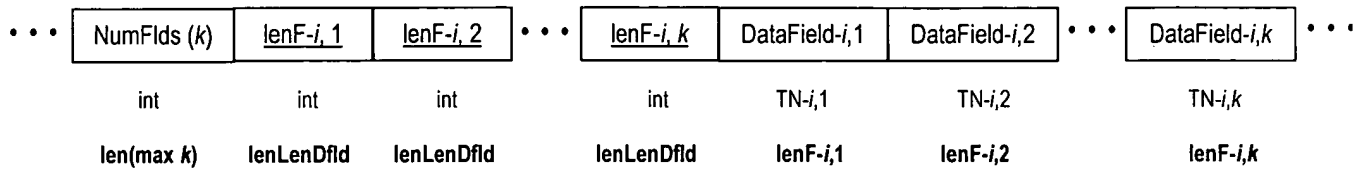
2) *FNF 0, FLF 1*, for some packet *i* (*1* ≤ *i* ≤ *j*):



3) *FNF 1, FLF 0 for some packet i:*



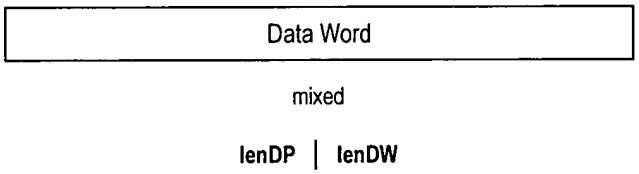
4) *FNF 0, FLF 0 for some packet i:*



Data words come in one four basic configurations, depending on whether the data-definition specifies:
1) fixed-length word
2) variable-length word, due to variable number of fields in some part
3) variable-length word, with fixed number of fields but variable field lengths
4) variable length word, with both variable number of fields and variable field lengths.

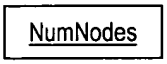
Note that in the latter three, the number of fields and/or the field-lengths must be stored in the actual data word. Note also that the condition where a variable number of fields in some part (packet) of the data-definition also does not have all fields of the same type, is forbidden in version 1.0 of NVSI, as it would require the storing of field types in the data-word as well. This level of complexity may be supported in subsequent versions.

Data-Word Registers (DWR-P & N)



The Data-Word registers (one in the PU and one in the NU) hold the contents of one data-word, to be stored into (DWR-P) or read from (DWR-N) a data-word in the DM (at the VNA specified in the associated IW). In 'compact' mode, the contents are loaded as one binary super-field (operand), either as read from memory, or as constructed from concatenating and encoding all component operands of the embedding-language variables into one bit-string. The parsing of the data-word after a read is done at the engine level, using the specifications contained in the corresponding DDW. The length of the register is specified either by the DDW (if the fdata-word is fixed-length), or the asociated IW (if variable length).

Number of Nodes Register (NNR)

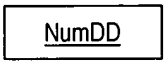


int

lenNC

The NNR holds the current total count (= *NumNodes*) of non-null nodes. This is used for iterating node-creation and population operations, and also for display by the NVSI *Console Monitor* (presented in the *NVSI System Developer* manual).

Data-Definition Count Register (DDCR)

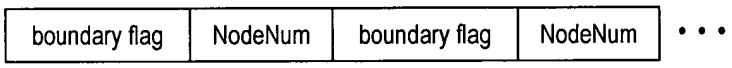


int

lenDDN

The DDCR holds the total count (= *NumDD*) of active data-definitions. This is used for DDA operations, and by the NVSI *Console Monitor*.

Node Free Index (NFI)



bln int • • •

2 **lenNC** • • •

The NFI stores the current *intervals* (inclusive range) of free (available, or 'null') nodes in the node Index Memory. Although a one-bit boundary flag is strictly sufficient to distinguish between a single node entry (for isolated free nodes), and an interval that requires two node-numbers to specify, a two-bit flag is used to provide for error-recovery in the event of a system crash, so that the NFI can be reconstructed. The <boundary flag> indicates one of three conditions that apply to the subsequent <NodeNum> field: `single-node`, `node-interval-start`, or `node-interval-end`. Thus, for example, the following would specify that the next free nodes are *17*, and then *23-28*:

• • • **11** *17* **01** *23* **10** *28* • • • ,

where **11** indicates an isolated free node to follow, **01** indicates the start of a range, and **10** indicates the end of a range.

Next Free Node Registers (NFNR 1 & 2)

NodeNum
int
lenNC

NFNR 1 & 2 are loaded – via the `Get-Free-Node-Range` instruction – from the Node-Free Index (NFI). Their primary use is for garbage collection, and for subsequent instantiation operations, to allow overwriting of unused Index words. The GFNR instruction is the only operation in the VM that invokes a significant sequence of embedded microcode, which searches the NFI for the first node-number greater than or equal to the value of the NC, and then loads the NFNR1 with that number, and NFNR2 with the end number of the range.

Field-Type Register (FTR) and Table (FTT)

TypeNum	Data-Word Field Type
int	char
len(NumTypes)	**lenTypeName**
	:
	: **NumTypes**

The FTT is a firmware table in the SCU that stores the list of preset data-word field types (tag, flag, character, integer, fixed-point, floating-point, function-pointer, node-pointer, and list-pointer). The FTR is a register in the SCU that is used to hold FTT words for storage or retrieval.

The FTT is extendible via SCU instructions from the *Configuration Engine.*

Index-segment Base-address Registers (IBR) and Stack

NodeNum	DDN-I	I/D Flag	DDN-D Flag	DDN-D
int	int	bln	bln	int
lenNC	**lenDDN**	**1**	**1**	**lenDDN**
		:		
		: **NumIBR**		

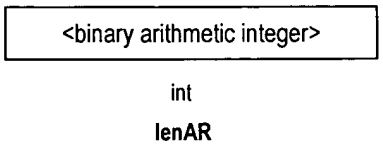
To provide for the segmenting of the solution-space into portions that share identical index and/or data-word configurations, but are distinct from other portions, the VM supports a simple indexed-addressing mechanism. Each IBR stores a base index address (node–number) that delineates the lower boundary of an index-memory segment that has common index-word definitions. Thus, any node whose address is greater than or equal to <NodeNum>, and is less than the node-number field of the *next* IBR, has the same configuration. The configuration of all the index words in the segment is given by the DDW pointed to by <DDN-I>.

The IBR stack consists of a set of **NumIBR** index base registers. If, at any point in the stack, the next IBR has a null <NodeNum> field, then the upper bound of the previous segment is taken to be *maxNodeNum* (that is, the end of Index Memory).

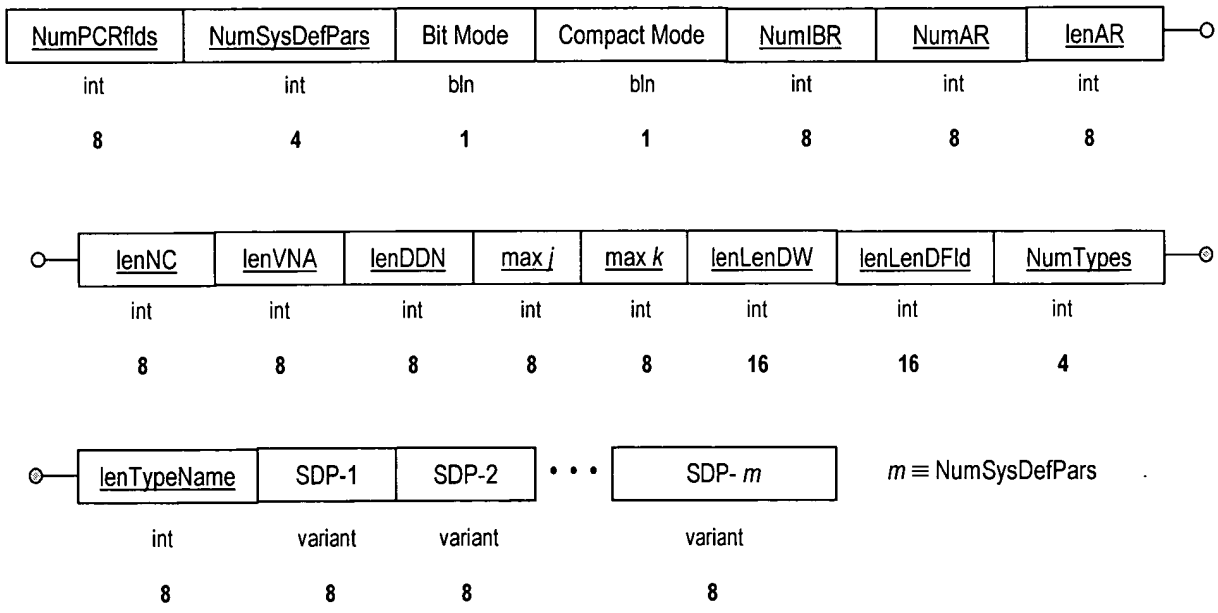
The <I/D Flag> determines whether (*1*) or not (*0*) the index words in that segment serve also as data-words. If so, then the subsequent fields in the IBR are ignored. If not, then all of the data words may share the same configuration (<DDN-D Flag> = *0*), which is then described by the DDW pointed to by <DDN-D>. This allows for matching homogeneous index and data segments. Otherwise (<DDN-D Flag> = *1*), the data-definition pointer for each data-word is stored in its respective index word (in the <DDN> field).

The IBR stack is loaded via the `set-index-base-register` instruction to the SCU.

Arithmetic Stack Register (AR)



Parameter Configuration Register (PCR)



The PCR stores the basic parameters that specify all the configurable architectural elements of the VM. These include the operating modes, field and register lengths (bit widths) for various components, and the number of registers or memory-words in the various configurable stacks, tables, and arrays. Detailed definitions for each parameter field are in Table 1.

Table 3. ***Virtual Machine Language (VML) Instruction Set***

VM Unit	Mnemonic	Operation	Operands	Opnd Type	Description
NC	SET	set NC	opnd.NC1	integer	Set NC to <opnd.NC1>.
	INC	increment NC	[opnd.NC2]	integer	Increment NC by one or <opnd.NC2>.
	DEC	decrement NC	[opnd.NC2]	integer	Decrement NC by one or <opnd.NC2>.
SCU	LPCR	load PCR	opnd.CU1 (*n*)	integer -array	Set each field in the PCR to its corresponding element in the integer-array <opnd.CU1>. The dimension *n* of CU1 is the number of fields in the PCR, given by <opnd.CU1(0)>→ PCR(0) ≡ NumPCRflds.
	LMA	load (set) IM & DM allocation registers	opnd.CU2 (2)	integer -array	Set the IMA register to the amount of memory, in megabytes, to be allocated to index-memory, as given by <opnd.CU2(0)>, and set the DMA register to the amount of memory allocation for data-memory, as given by <opnd.CU2(1)>.
	LDDR	load DDR	opnd.CU3 (*n*)	variant -array	Set each field in the DDR to its corresponding element in the variant-array <opnd.CU3>. The dimension *n* of CU3 is available from the configuration details of the data definition. If *n* = *1* (one element), then the single operand field is loaded into DDR(0) to specify the DDN for a subsequent read (*GDDW*) instruction.
	SDDW	store DDA word	[opnd.CU4]	boolean	Store (write to) each field in the DDA memory-word pointed to by DDN = DDR(0) with its corresponding field value in the DDR. If the DDN is new (not found in the DDA), then increment DDCR. If <opnd.CU4> = True, then the instruction becomes a *delete*, and the DDN of the corresponding DDA word is set to zero (which flags the word as null), and the DDCR is then decremented.
	GDDW	get (read) DDA word	-	-	Set each field in the DDR to the corresponding field in the DDA word pointed to by DDN = DDR(0).
	CDDA	clear DDA	-	-	Set the DDN field of all DDA words (count given by DDCR) to zero. Set DDCR to zero. This happens by default at "power-up" (initialization).
	SFT	store Field Type word	opnd.CU5 (2)	variant -array	Load the FTR from <opnd.CU5>, then execute a store into FTT word pointed to by FTR(0) = CU5(0) ≡ TypeNum. If <opnd.CU5> is a single element, then only FTR(0) is loaded, and no store is executed, in preparation for a subsequent read (*GFT*) instruction.
	GFT	get (read) FT word	-	-	Load the FTR from the FTT word pointed to by FTR(0) ≡ TypeNum.
	SIBR	set an IB Register	opnd.CU6 opnd.CU7 (5)	integer variant -array	Set all fields of IBR(*i*), where *i* = <opnd.CU6>, to <opnd.CU7>.

Table 3 (cont'd). ***Virtual Machine Language (VML) Instruction Set***

IU	RIW	read IW	-	-	Set IWR to IW(NC). This allows the reading of IW contents, primarily as a precursor to navigating, populating or re-instantiating a node.
	LIWR	load IWR	opnd.I1 (*n*)	variant -array	If the VM is operating in 'compact mode', then the single (*n = 1*) operand bit-string is loaded into the IWR (Index Word portion). Otherwise, the fields of the IWR, as defined (prior) by the configuration of the DDW pointed to by the DDN-I field of the relevant IBR (based upon NC), are loaded from the corresponding operand-array elements. The length of the IWR, and the number of fields (the dimension *n* of the operand array), are calculated prior to this instruction, from details contained in the associated DDW. This instruction is typically a precursor to executing a *CRE* instruction.
	CRE	create node (store IW)	[opnd.I2]	boolean	If <opnd.I2> = True, then create (*instantiate*) a node at node-number = NNR, else create node at node-number = NC (default). To create a node, the contents of the IWR (except the Null Flag), are stored as an IW in Index Memory at relative address = node-number. In '*compact mode*', this is a simple binary transfer. Otherwise, the fields of the IWR are stored into corresponding IW-array elements. The NC is then incremented (to optimize serial *instantiation*). If the node had been null, then clear the Null Flag of the IW, update the NFI, and increment NNR. Note that if the IW is a *data version*, then the ADFs may be null, to be filled later via a *PDW* instruction.
	DEL	delete node	-	-	Delete node (set Null Flag) in IW at node-number = NC. Also, if node had not been null, then update NFI, and decrement NNR.
	GFNR	get free node range	-	-	Sets NFNR1 & NFNR2 to the boundaries of the next-free-node-range, where NFNR1 ≥ NC. (The internal IM controller scans the NFI, and finds the first null node or node-range ≥ NC. This instruction is therefore slower, as it executes an embedded microcode procedure.)

Table 3 (cont'd). *Virtual Machine Language (VML) Instruction Set*

PU	LDWR	load DWR	opnd.P1 (*n*)	variant -array	If the VM is operating in '*compact mode*', then the single (*n = 1*) operand bit-string is loaded into DWR-P. Otherwise, the fields of DWR-P, as defined (prior) by the configuration of the DDW pointed to by either the DDN-D field of the relevant IBR (based upon NC), or by the DDN of the associated IW, are loaded from the corresponding operand-array elements. The length of the DWR, and the number of fields (the dimension *n* of the operand array), are calculated prior to this instruction, from details contained in the associated DDW.
	PDW	populate (store) DW	[opnd.P2] [opnd.P3]	boolean boolean	If <opnd.P2> = True, the IW is a *data-version* and the contents of DWR-P are stored into the data portion of the IW located at address = NC. Otherwise (*index version*), store the contents of DWR-P into the DW located at: • if <opnd.P3> = False (pre-execution address retrieval), then VNA ← IWR(VNA); • else (execution-time address retrieval), VNA ← IW(NC). Note that if the length of the DW has changed, then a new VNA must be obtained by the DM internal controller (in conjunction with the PMC), which then automatically rewrites the new VNA into the corresponding field of both the IW, and the IWR In '*compact mode*', the store operation is a simple binary transfer. Otherwise, the fields of the DWR are stored into corresponding DW (or IW) array elements. The NC is then incremented (to optimize serial *population*).
NU	GDW	get (read) DW	[opnd.N1]	boolean	Load DWR-N with the contents of the DW at the VNA selected by <opnd.N1>, same action as for <opnd.P3>. *GDW* is the basic navigation operation.

key:

[ ]	optional operand
< >	operand name in text
≡	'is defined as'
←	'takes contents from'
→	'sends contents to'

